Kam Python shranjuje stvari

V rednem delu teh predavanj se igramo z raznimi čudnimi stvarmi, da bi pokazali razliko med imenom reči in rečjo samo. Isti stvari smo dali dve imeni in videli, da se, če jo spreminjamo prek enega imena, seveda spreminja tudi tisto, kar vidimo pod drugim imenom (to je, drugim imenom za isto reč). Če je neobrit Janez, je neobrit tudi Demšar.

Povedali smo tudi, kaj so v resnici argumenti funkcij: vedejo se kot *lokalna* imena za objekte, ki smo jih podali funkciji kot argument. Ničesar pa nismo povedali o tem, kje je vse to shranjeno.

Python shranjuje imena in vrednosti - kako priročno! - kar v slovarju, ki si ga tiho sestavi v ta namen. Ključi tega slovarja so imena, vrednosti pa pač objekti, na katere se ta imena nanašajo. Tega slovarja na skriva: do njega pridemo, recimo, prek funkcije `globals()`.

```
globals()
```

```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['', 'globals()'],
 '_oh': {},
 '_dh': ['/Users/janez/Desktop/predavanja/p1/dodatna predavanja'],
 'In': ['', 'globals()'],
 'Out': {},
 'get_ipython': <bound method InteractiveShell.get_ipython of <ipykernel.zmqshell.ZMQIntera
 'exit': <IPython.core.autocall.ZMQExitAutocall at 0x7f9da3f937d0>,
 'quit': <IPython.core.autocall.ZMQExitAutocall at 0x7f9da3f937d0>,
 '_': '',
 '__': '',
 '___': '',
 '_i': '',
 '_ii': '',
 '_iii': '',
 '_i1': 'globals()'}
```

`globals()` nam torej vrne slovar z imeni in vrednostmi spremenljivk. Ta že v začetku ni prazen. Nekaj te šare vedno naredi Python, še veliko več pa je doda Jupyter Notebook, v katerem vse tole predvajamo. Da ne gledamo vsega tega, si zapomnimo, kaj imamo tu in poslej bomo izpisovali samo, kar se pojavi na novo.

```
stuff = set(globals()) | {"stuff"}
```

Sestavimo dve spremenljivki in izpišimo vsebino `globals()` (brez `stuff` in vsega, kar se začne s `_`, saj Jupyter dodaja vedno več in več tega).

```
x = 12
ime = "Benjamin"
```

```
{k: v for k, v in globals().items() if k not in stuff and k[0] != "_"}
```

```
{'x': 12, 'ime': 'Benjamin'}
```

Ko napišemo `x = 12`, se v slovarju pojavi nov element; ključ je niz `"x"`, vrednost pa 12. Podobno se zgodi, ko napišemo `ime = "Benjamin"`.

Recimo, da zdaj pa vtipkajmo `x + 7`. Python kot vemo, najprej preveri, ali obstaja spremenljivka z imenom `x`. To stori preprosto tako, da pogleda v tale slovar. Če obstaja ključ `x`, v izrazu `x + 7` uporabi pripadajočo vrednost. Če ga ni, javi napako.

Napišem lahko tudi `del x`, kar pobriše spremenljivko (ups, ime!) `x`. O tem se nismo posebej pogovarjali, ker ni bilo potrebno. Če bi se pogovarjali o tem, pa bi vam lahko zdajle povedal, da `del x` ne naredi nič drugega, kot da pobriše dotični ključ (in, seveda, pripadajočo vrednost) iz slovarja. In v tem primeru bi to tudi pokazal.

```
del x
```

```
{k: v for k, v in globals().items() if k not in stuff and k[0] != "_"}
```

```
{'ime': 'Benjamin'}
```

```
spremenljivke = globals()
```

```
{k: v for k, v in spremenljivke.items() if k not in stuff and k[0] != "_" and k != "spremenl
```

```
{'ime': 'Benjamin'}
```

Zdaj se ime `spremenljivke` nanaša na slovar, ki ga vrne `globals()`. Ta slovar je slovar spremenljivk in zato, he he, vsebuje tudi ključ `spremenljivke`, pripadajoča vrednost pa je kar ta slovar sam. Vendar ga raje nismo izpisali, ker bi se izpisala tudi vsa Jupytrova šara, zato smo dodali še `k != spremenljivke`).

Če je `spremenljivke` torej v resnici slovar spremenljivk - lahko vanj dodajamo nove spremenljivke?

```
spremenljivke["foo"] = 42
spremenljivke["bar"] = [1, 2, 3]
```

Pa preverimo.

```
foo
```

```
42
```

```
bar
```

```
[1, 2, 3]
```

Če torej napišemo `foo = 42` (kot bi počeli normalno) je to samo bližnjica za `globals()["a"] = 42` (če smo natančni, bližnjica za `globals().__setitem__("a", 42)`, ampak tako daleč v drobovje Pythona ne bomo rinili).

Podobno je `x + 7` samo bližnjica za `globals()["x"] + 7`. V resnici se prvo "prevede" v drugo (ali, če smo natančni, v nekaj podobnega drugemu).

Kar smo pravkar videli, ni posebej uporabno. Tu je, kar pač Python tega ne skriva. Ni pa mišljeno, da bi se to uporabljalo. Tudi mi smo pogledali samo zato, da smo videli, kako Python v resnici deluje. To nam bo namreč pomagalo razumeti, kar sledi.

Lokalne shrambe

Vemo, da ima vsaka funkcija svoje spremenljivke, svoja imena. Kako to deluje, najbrž slutimo: vsaka ima svoj slovar. Do njega ne pridemo prek `globals()` temveč prek `locals()`.

```
def f(x, y):
    z = 13
    print(locals())

f(1, "Benjamin")
```

```
{'x': 1, 'y': 'Benjamin', 'z': 13}
```

Lokalne spremenljivke, ki jih pozna `f`, so torej `x` in `y`, ki ju je dobila kot argument, in `z`, ki jo je ustvarila kasneje.

Poleg tega imajo funkcije seveda tudi dostop do globalnih spremenljivk. Predstavljamo si, da pogledajo v oba slovarja, najprej v `locals`, nato še v `globals`.

Predstavljamo si lahko? Da. Predstavljamo. Slovar `globals()` je resničen, `locals()` je fake. Python ga sestavi, če ravno prosimo zanj. Če ga spreminjamo, pa Python ne bo v resnici dodajal spremenljivk. Lokalne spremenljivke funkcije niso shranjene v slovarju, temveč na fiksnih lokacijah na skladu. Tule (še) ne bomo šli pregloboko v to, kako delujejo funkcije, le nakažimo, za kaj gre.

Ko takole kramljam s Pythonom in si izmišljam spremenljivke, recimo seznam `imena`, ki ga vedno znova definiram na predavanjih, Python nikoli ne ve, katero spremenljivko si bom izmislil v naslednjem trenutku. Ker pač to počnem sproti. S funkcijami pa ni tako. Tam si spremenljivk ne izmišljam sproti. No, seveda, ko programiram, jih seveda sestavljam sproti, med tipkanjem, brez deklariranja. Vendar Python takrat funkcije še ne izvaja. Preden jo pokličem, pa Python že vidi vso funkcijo (in jo v resnici celo prevede, čisto tako kot Java v JVM, C# v CLI in Javascript v nimam pojma kaj). Takrat ve tudi za vse njene spremenljivke; ker ne bo nihče spreminjal kode funkcije, tudi nihče ne bo več

dodajal spremenljivk. (Če bo kdo spreminjal funkcijo, pa se bo itak ponovno prevedla.)

Funkcija zato "ve", koliko spremenljivk ima in kakšna so njihova imena. Gornja funkcija `f` ima tri spremenljivke in njihova imena so `x`, `y` in `z` - shranjena v terkah, da jih lahko vidimo.

```
f.__code__.co_nlocals
```

```
3
```

```
f.__code__.co_varnames
```

```
('x', 'y', 'z')
```

Zakaj tako? Ker je hitrejše. Veliko hitrejše. Zato so v Pythonu lokalne spremenljivke hitrejše od globalnih, ker ne zahtevajo (sicer hitrega) pogleda v slovar.

Kot zanimivost: tudi katere konstante bo potrebovala funkcija, vemo vnaprej, zato ni potrebno, da bi `13` v gornji funkciji ustvarili v vsakem klicu posebej. Pripravi jo že prevajalnik, skupaj s konstantno `None`, ki jo funkcija vrne kot rezultat.

```
f.__code__.co_consts
```

```
(None, 13)
```

Zakar mi koristi vedeti vse to?

Nimam pojma. Ker je zanimivo?

Imenski prostori modulov

Kje so shranjene spremenljivke modulov?

Najbrž v nekem slovarju. Globalnem ali lokalnem? V globalnem - očitno ne. V lokalnem? Ta pa ne obstaja. Vse spremenljivke funkcije so na skladu.

Polovico zgodbe vidimo tu:

```
import math
math.__dict__
```

```
{'__name__': 'math',
 '__doc__': 'This module provides access to the mathematical functions\ndefined by the C sta
 '__package__': '',
 '__loader__': <_frozen_importlib_external.ExtensionFileLoader at 0x7f9da1c89290>,
 '__spec__': ModuleSpec(name='math', loader=<_frozen_importlib_external.ExtensionFileLoader
 'acos': <function math.acos(x, /)>,
 'acosh': <function math.acosh(x, /)>,
 'asin': <function math.asin(x, /)>,
 'asinh': <function math.asinh(x, /)>,
```

```
'atan': <function math.atan(x, /)>,
'atan2': <function math.atan2(y, x, /)>,
'atanh': <function math.atanh(x, /)>,
'ceil': <function math.ceil(x, /)>,
'copysign': <function math.copysign(x, y, /)>,
'cos': <function math.cos(x, /)>,
'cosh': <function math.cosh(x, /)>,
'degrees': <function math.degrees(x, /)>,
'erf': <function math.erf(x, /)>,
'erfc': <function math.erfc(x, /)>,
'exp': <function math.exp(x, /)>,
'expm1': <function math.expm1(x, /)>,
'fabs': <function math.fabs(x, /)>,
'factorial': <function math.factorial(x, /)>,
'floor': <function math.floor(x, /)>,
'fmod': <function math.fmod(x, y, /)>,
'frexp': <function math.frexp(x, /)>,
'fsum': <function math.fsum(seq, /)>,
'gamma': <function math.gamma(x, /)>,
'gcd': <function math.gcd(x, y, /)>,
'hypot': <function math.hypot(x, y, /)>,
'isclose': <function math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)>,
'isfinite': <function math.isfinite(x, /)>,
'isinf': <function math.isinf(x, /)>,
'isnan': <function math.isnan(x, /)>,
'ldexp': <function math.ldexp(x, i, /)>,
'lgamma': <function math.lgamma(x, /)>,
'log': <function math.log>,
'log1p': <function math.log1p(x, /)>,
'log10': <function math.log10(x, /)>,
'log2': <function math.log2(x, /)>,
'modf': <function math.modf(x, /)>,
'pow': <function math.pow(x, y, /)>,
'radians': <function math.radians(x, /)>,
'remainder': <function math.remainder(x, y, /)>,
'sin': <function math.sin(x, /)>,
'sinh': <function math.sinh(x, /)>,
'sqrt': <function math.sqrt(x, /)>,
'tan': <function math.tan(x, /)>,
'tanh': <function math.tanh(x, /)>,
'trunc': <function math.trunc(x, /)>,
'pi': 3.141592653589793,
'e': 2.718281828459045,
'tau': 6.283185307179586,
'inf': inf,
'nan': nan,
```

```
 '__file__': '/Users/janez/miniconda3/envs/prog/lib/python3.7/lib-dynload/math.cpython-37m-d
```

Vse, kar je v modulu, je shranjeno v modulovem slovarju `__dict__`. Pripravil sem modul `primer.py`, ki je videti tako.

```
x = 15

def f(y):
    print(globals())
    return x * y

def g():
    return f(12)
```

Uvozimo ga in preverimo, kaj je v njegovem `__dict__`.

```
import primer

# __dict__ vsebuje __builtins__, ki vsebuje `print` in `globals` in druge funkcije
# Ker jih je veliko, jih nočemo izpisati: naredimo kopijo slovarja, pobrišimo
# `__builtins__ in ga izpišimo
primdict = primer.__dict__.copy()
del primdict["__builtins__"]

primdict
```

```
{'__name__': 'primer',
 '__doc__': None,
 '__package__': '',
 '__loader__': <_frozen_importlib_external.SourceFileLoader at 0x7f9da58b2e10>,
 '__spec__': ModuleSpec(name='primer', loader=<_frozen_importlib_external.SourceFileLoader o
 '__file__': '/Users/janez/Desktop/predavanja/p1/dodatna predavanja/primer.py',
 '__cached__': '/Users/janez/Desktop/predavanja/p1/dodatna predavanja/__pycache__/primer.cpy
 'x': 15,
 'f': <function primer.f(y)>,
 'g': <function primer.g()>}
```

Po pričakovanjih.

A z vidika funkcije `f` je `x` očitno globalna spremenljivka. Preverimo, če je res - pokličimo funkcijo `f`, pa nam bo izpisala globalne spremenljivke.

```
primer.f(12)
```

```
{'__name__': 'primer', '__doc__': None, '__package__': '', '__loader__': <_frozen_importlib_
```

```
Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved., 'credits':     Thanks to CWI, CNRI, BeOpen.com, Zope Corporation and a
    for supporting Python development.  See www.python.org for more information., 'license':
```

```
180
```

Izpisalo se je še vse živo (vsebina tistega `__builtins__`). Bistveno je na koncu: `x`, `f` in `g`. Če modul `primer` vpraša po `globals()`, dobi nekaj drugega, kot če po `globals()` vprašamo v svojem programu. Vsak modul ima svoj `globals()`.

Tako mora biti. Funkcija `g` kliče funkcijo `f`. Pokliče jo kot `f`, ne `primer.f`. Zato mora biti med globalnimi spremenljivkami. Če bi hotela funkcija `g` klicati `primer.f`, potem bi morala prej uvoziti modul `primer`, torej samega sebe. To se sicer da, vendar se lahko konča slabo, če ne pazimo, kako to storimo. Poleg tega tega nikoli ne počnemo, ker ni smiselno.

import in import from

Zgoraj smo napisali `import math`. Modul `math` se pojavi med globalnimi spremenljivkami.

```
globals()["math"]
```

```
<module 'math' from '/Users/janez/miniconda3/envs/prog/lib/python3.7/lib-dynload/math.cpytho
```

Če namesto tega napišemo `from math import cos`, se med globalnimi spremenljivkami pojavi `cos`.

```
from math import cos

globals()["cos"]
```

```
<function math.cos(x, /)>
```

__name__

Ena od globalnih spremenljivk, ki se je pojavila sama od sebe, je `__name__`. Ta je imela doslej vrednost `__main__`.

```
__name__
```

```
'__main__'
```

Poglejte, kakšno vrednost ima v globalnih spremenljivkah modula `primer`:

```
primer.__name__
```

```
'primer'
```

Pod takim imenom se pojavi tudi v modulu `primer`, če bi ga, recimo, izpisali ali preverjali v `primer`. Na ta način lahko program ve, ali ga izvajamo kot program ali kot modul.

Na koncu testov, ki jih dobivate ob domačih nalogah, vedno pišem

```
if __name__ == "__main__":
    unittest.main()
```

To požene teste, vendar le, če program poženemo kot običajen program. Če pa PyCharm prepozna, da so v njem testi, ga bo uvozil kot modul, da bo potem lahko dostopal do razredov s testi. V tem primeru `__name__` ne bo enak `__main__`, temveč imenu datoteke in funkcija `unittest.main()` se ne bo poklicala. Tako mora biti, da bo teste poklical PyCharm.

Imenski prostori razredov

Če bi že poznali razrede, bi nas zanimalo, kje so shranjeni atributi. Prav tako v slovarju in ime mu je, tako kot pri modulih, `__dict__`.

```
class A:
    def __init__(self):
        self.x = 42

    def metoda(self):
        return "Benjamin"

a = A()
a.x
```

```
42
```

```
a.__dict__
```

```
{'x': 42}
```

```
a.foo = 13

a.__dict__
```

```
{'x': 42, 'foo': 13}
```

```
a.__dict__["bar"] = "Cilka"

a.bar
```

```
'Cilka'
```

Vse po pričakovanjih.

Kaj pa to?

```
a.metoda()
```

```
'Benjamin'
```

Odkod `metoda`, če je vendar ni v `a.__dict__`? Ta je pa tu.

```
A.__dict__
```

```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.A.__init__(self)>,
              'metoda': <function __main__.A.metoda(self)>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '__doc__': None})
```

A s tem, kako razredi iščejo metode in atribute se bomo raje ukvarjali, ko pridemo do razredov.